Formally Verifying Data and Control with Weak Reachability Invariants

Jeffrey Su, David L. Dill, and Jens U. Skakkebæk

Computer Systems Laboratory,
Stanford University, Stanford, CA 94305, USA
Phone: (650) 725-9046, Fax: (650) 725-6949
E-mail: {xsu, dill, jus}@cs.stanford.edu

Abstract. Existing formal verification methods do not handle systems that combine state machines and data paths very well. Model checking deals with finite-state machines efficiently, but model checking full designs is infeasible because of the large amount of state in the data path. Theorem-proving methods may be effective for verifying data path operations, but verifying the control requires finding and proving inductive invariants that characterize the reachable states of the system.

We present a new approach to verification of systems that combine control FSMs and data path operations. Invariants are specified only for a small set of control states, called *clean states*, where the invariants are especially simple. We avoid the need to specify the invariants for the unclean states by symbolically simulating over all paths to find the possible next clean states.

The set of all paths from one clean state to the next is represented by a regular expression, which is extracted from the control FSMs. The number of paths is infinite only if the regular expression contains stars. The method uses a heuristic to generalize the symbolic state to cover all of the paths of the starred expression. We have implemented a prototype tool for guiding an existing symbolic simulator and verification tool and used it successfully to prove properties of the Instruction Fetch Unit of TORCH, a superscalar microprocessor designed at Stanford. With much less effort, we were able to find all the bugs in the unit that were found earlier by manually strengthening the invariants.

1 Introduction

Existing formal verification methods do not handle systems that combine finite-state machines (FSMs) and data paths very well. Model checking [6, 5, 4] the full design is infeasible because of the large amount of state in the data path. Verifying the control FSMs in isolation is difficult, because specifying them independently is difficult – the design requirements are usually stated as properties of the data path, not the FSMs themselves. The specification of the control is that it causes the data path property to be satisfied. Abstracting the data path to reduce the amount of state is sometimes possible, but it is subtle and may require changes in the control that introduce false errors or cause true errors to be missed.

Theorem-proving methods require finding and proving inductive invariants that characterize the reachable states of the system. This is not necessarily difficult for a system that is implementing an algorithm (e.g. a floating point unit). However, when a design has significant control complexity, finding invariants is primarily a tedious manual trial-and-error process.

One path to a solution to these problems would be to find ways to reduce the effort to find inductive invariants in these designs, through automation or methodology. [1] Although the problem of automatic invariant discovery has been studied over the years, there is not yet a complete solution to the problem [13, 11, 7, 3, 18, 2, 1]. In particular most of the work seems not to be applicable to register transfer level (RTL) hardware designs. Most current designs are described at RTL using a hardware description language (HDL) such as Verilog or VHDL, and are then manually or automatically synthesized.

Some of the invariants that are needed in a proof are *historyless* properties, by which we mean that they are provable with no assumptions about the previous state of the system. Equivalently, a historyless property is true of every state that has at least one predecessor in a state transition graph of the system behavior. The concept can be extended to include properties that hold for all states with at least one k-predecessor, where a k-predecessor is a state from which there is a path of length k to the state satisfying the invariants. Of course, for a historyless property to be an invariant, the initial state must satisfy the property. In RTL designs, historyless invariants are surprisingly useful, because they capture some important properties of data propagating through acyclic chains of registers. Also, multi-phase designs (where alternating layers of registers are clocked on different phases of a single clock) tend to lead to historyless invariants that relate the contents of consecutive latches which are clocked in different phases. The discovery and use of historyless invariants in RTL designs was explored in this conference in 1996 [24]. The discovery of historyless properties is also a component of the work cited above for finding invariants in software and protocol descriptions.

This paper attacks the invariant problem in another, complementary, way, by trying to simplify the problem. Examination of a number of designs has revealed a general tendency that can be exploited. Many systems can be thought of as processing a sequence of transactions, where processing a transaction involves a sequence of steps. When the system is not processing a transaction, we say it is in a *clean state*. This paper is based on the observation that *the invariants that needed for the clean states are much simpler than for the other states.* The reason for this is simple: much of the complexity of inductive invariants stems from capturing the bookkeeping that happens during the processing of a transaction.

The partial solution proposed here is to identify the clean states of the system and specify their invariants. These invariants are proved by symbolically simulating along every path from each clean state q to the next clean state q', and showing that if the invariant held in q, it will hold in each q' no matter what path was taken from q to q'.

[1] It is important to distinguish between the difficulties of *finding inductive invariants* vs. *proving inductive invariants*. In general, finding the invariants is much more difficult than proving them after they have been found.

The paths between the clean states are described using regular expressions.

The most serious technical difficulty is that there can be an infinite number of paths from q to q', because of cycles of unclean states along the path. However, in some systems at least, these cycles are simple wait loops, so it can be shown that paths that go around the cycle any number of times are equivalent to those that go around zero or one times (these ideas are made more precise below).

Viewed at the level of abstraction of the previous paragraph, there is little new about this approach. Indeed, it is very similar to very early work on program verification, especially the inductive assertions method of Floyd [9], which cuts all cycles in a program flow graph, then finds assertions that hold at the end of the cycle if they hold at the beginning. King specifically used symbolic simulation was to derive invariants [14]. Symbolic simulation along paths between major states has also applied to formal verification of microprograms [8, 15, 16]. The idea of using regular expressions to represent all possible execution paths comes directly from Tarjan [25], who suggested using regular algebra for program flow analysis.

However, RTL hardware design is quite different from sequential program and microprogram verification. To a programmer, RTL designs would appear to be very low-level. Control flow is encoded into one or several FSMs which are separated from the data path. Second, symbolic simulation of even one step results in a huge expression for the symbolic state, since hundreds of state variables may be updated simultaneously. In contrast, a single step in a sequential program or microprogram would typically be a small number of assignments to variables.

While the approach comes out of a tradition of program verification and analysis, these ideas have not previously been applied to RTL designs, however. The reason for this is probably that synthesizable HDL descriptions do not express control flow in the same way as sequential programs. Instead, FSM controllers are defined which are separate from the data path. The method proposed here extracts the regular expressions from the FSMs in the design, not the syntactic structure of the HDL. The other new insight is that, in many cases, finding an invariant around a loop *between clean states* is simple, because the loop often represents a wait state.

These results are preliminary. The proofs still require more effort than one would hope, the invariants are still large (but much smaller than without the method), and it has only been evaluated on one real design. However, it is a new approach that appears to have the potential to be a practical verification method for some designs that are difficult or impossible by other methods.

A simple example

A very simple example is depicted in Figure 1, which is used to make some of the above discussion more concrete. The example consists of three registers, controlled by a small state machine. Periodically, the *new_data* input to the state machine goes high, and, in the next cycle, a new value is loaded into *source*. The state machine then waits for a *ready* signal indicating that the new value can be transferred to *dest*. Then, half the data in *source* is copied to *middle* and the state machine enters state T_1; in the next cycle, that value is copied to *dest*; simultaneously, the remaining data is copied from

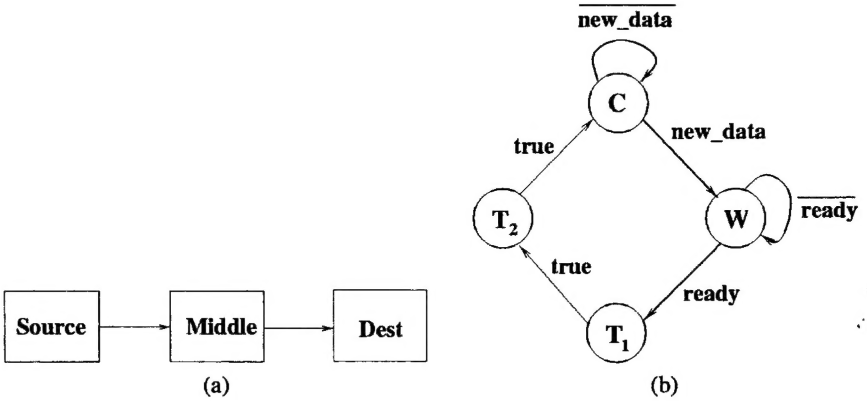

Fig. 1. (a) Data Path (b) Control FSM

source to *middle* and the state machine enters T_2. In the next cycle, the remaining data is copied from *middle* to *dest* and the state machine returns to state C.

Suppose the system is specified to have the property that, whenever it is in state C, the contents of *source* and *dest* registers are the same. Proving this by induction would require providing an inductive invariant (which is actually the induction hypothesis), which must be preserved by all state transitions. The inductive invariant would need to include properties in addition to the basic requirement: When in state T_1, the contents of *middle* equals the lower half of *source*, and when in T_2, the contents of *middle* equals the upper half of *source* and the contents of *dest* equals the lower half of *source*. Intuitively, the inductive invariant needs to track data movement step-by-step in its journey from *source* to *destination.*

This type of phenomenon occurs very frequently, but the systems are obviously more complex than this example, as are the additional properties required of the inductive invariants. In real memory systems, for instance, memory values may be loaded in many cycles and pass through several intermediate registers while being packed together in the right form. An invariant must relate all the intermediate registers with the memory and cache for all the different modes of execution. Hence, in practice, finding invariants is by far the most time-consuming task when verifying with a theorem prover.

The verification of a property only in clean states is illustrated in Figure 2.

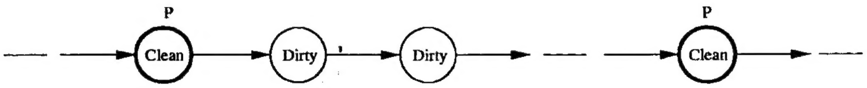

Fig. 2. Verifying a property P only in the clean states.

The symbolic simulator operates on *symbolic states*, which map state variables to logical expressions. For each clean state, the simulator is presented with a symbolic state that initializes all storage elements to distinct symbolic constants. Also, a sequence of logical conditions on the data path and inputs must be satisfied for a particular path to be followed through the FSM. The conjunction of these conditions is called the *path constraint*. The result of symbolically simulating over a path is a new symbolic state and a path constraint.

To prove the invariant, it is then necessary to show that for each clean state that, if the path constraint is satisfied and the invariant holds on the initial symbolic state, it also holds for the symbolic states reached by symbolically simulating over all paths. The set of all paths from one clean state to the next is represented by a regular expression, which is extracted from the control FSMs. There number of paths is infinite only if the regular expression contains stars. The method uses a heuristic to generalize the symbolic state to cover all of the paths of the starred expression. From another viewpoint, the method compresses sequences of steps through the state graph so that there is only a single composite step from each clean state to the next. The composite step is computed by symbolically simulating along multi-step paths.

In the example of Figure 1, we would define one clean state, C. Only the initial invariant, that the *source* and *dest* registers are equal when in state C, would need to be proved, but it would need to be proved for the symbolic states yielded by simulating over all paths from C back to C.

This method has been used to prove invariants for the Instruction Fetch Unit of TORCH [22, 21], a superscalar microprocessor designed at Stanford. The same bugs were found as in an earlier effort [23], but with a major reduction in effort.

2 The Verification Method

Extracting regular expressions for control paths

Regular expressions are used because they make it easy to identify and handle cycles in the FSMs. Hence, the first step of the method is to obtain regular expressions describing all paths between the clean states. This requires extracting the state machine controllers in the design, constructing a single product machine (called "the FSM," below), locating the clean states within the FSM, and, for each clean state, deriving a regular expression describing the control paths to the next clean states.

Currently, all of these steps are manual, although everything can be done by well-known algorithms, except extraction of the FSMs from the HDL description. FSM extraction can probably be done automatically in many cases; however, for the designs we are considering, manual extraction by the designer is not difficult.

The FSM is a high-level finite state machine. The outputs are not modelled, since they are not relevant for invariant checking. The transitions are labelled with logical formulas, which are written in a quantifier-free fragment of first-order logic which includes Boolean signals and operators, uninterpreted functions, equality, bitvectors, arithmetic, and arrays. The logical formulas can include individual Boolean signals (which appear as propositional symbols), or predicates on the data path signals (e.g., "$r_1 = r_2$" to represent the output of a comparator between two registers).

The alphabet of the regular language consists of the set of input values to the FSM. Sets of inputs are represented as logical formulas (taken directly from the FSM).

More formally, a regular expression consists of

- The empty string, ϵ;
- A Boolean formula over the input signals to the FSM;
- A concatenation of two regular expressions, $(\alpha \cdot \beta)$;
- A union of two regular expressions, $(\alpha + \beta)$;
- Kleene closure of a regular expression, α^*.

In our examples, we assume that * has higher precedence than $\cdot$, which has higher precedence than $+$, and drop parenthesis accordingly. In the Lisp implementation, the regular expressions are actually represented using Lisp syntax.

For every clean state, we can construct from the original FSM a finite automaton describing the set of all paths of non-zero length from that clean state to another clean state that do not have clean states except at the beginning and end. The regular expression of all input sequences accepted by this finite automaton can be computed by standard algorithms from finite automaton theory (see [10], for example).

In the example of Figure 1, the desired regular expression is:

$$\overline{new\_data} + new\_data \cdot \overline{ready}^* \cdot ready \cdot True \cdot True$$

In this case, the Boolean combinations are all single signals or their complements (indicated by the overlining). In general, of course, a single "symbol" in the regular expression may be a more complex Boolean expression.

Guiding symbolic simulation

The symbolic simulator simulates a high-level netlist (HLN), which is a graph structure representing a digital circuit. The vertices of the graph are circuit elements, such as adders, Boolean gates, registers, and memories. The edges in the graph represent arrays of wires. A *symbolic state* is a map from state variables (registers and memories) to logical expressions representing the symbolic values of the state variables. Given an HLN, a symbolic state S, and logical formulas for the circuit inputs, the symbolic simulator returns a symbolic state representing the updated values of the state variables after one clock cycle of execution of the circuit.

Symbolically simulating along the paths of a regular expression is called *path simulation*. Path simulation operates on pairs $\langle S, P \rangle$, where S is a symbolic state and P is a conjunction of Boolean formulas, called a *path constraint*. Given an HLN, a pair $\langle S, P \rangle$, path simulation produces a finite set of pairs $\{\langle S', P' \rangle\}$, where P' is the conjunction of P with additional path constraints.

For each clean state, we start with a symbolic state S that assigns the correct constants to the FSM state variables and distinct symbolic constants to all other state variables. Path simulation along the regular expression from the clean state to every next clean state yields a set of $\langle S', P' \rangle$ pairs representing the symbolic states and path constraints when the next clean states are reached.

If I is a Boolean formula on the state variables, $I(S)$ represents the formula obtained by substituting for each state variable in I the corresponding logical formula from S. To prove that I is an invariant, we must show that $P' \wedge I(S) \Rightarrow I(S')$ for each pair produced by the path simulation. The assumption that P' holds is justified, since the FSM could only have followed the path if all the conditions in P' were satisfied.

At times, it is necessary or desirable to approximate a set of pairs with a single pair. We say $\langle S', P' \rangle$ *approximates* $\langle S'', P'' \rangle$ if the validity of $P' \wedge I(S) \Rightarrow I(S')$ is a sufficient condition for the validity of $P'' \wedge I(S) \Rightarrow I(S'')$. The approximation is *conservative*: The approximation may cause the proof of a valid invariant to fail, but will never allow an invalid invariant to be proved.

A simple approximation is used, called the *merge* of a set of pairs into a single pair. If both states in the pair map a state variable to the same logical formula, the merged state maps it to the same formula; otherwise, the merged state maps the state variable to a *fresh symbolic constant*, which is a named constant that has not previously appeared in a symbolic state or path predicate. The merge of two pairs $\langle S, P \rangle$ and $\langle S', P' \rangle$ is a pair $\langle S'', P'' \rangle$, where S'' is the merge of S and S', and P'' is the disjunction of P and P'. Since validity is, by definition, truth in all interpretations, the fresh variables are implicitly universally quantified, so this approximation satisfies the definition above.

Path simulation is guided by the recursive structure of the regular expression. Henceforth, "simulate means "symbolic simulate."

Concatenation Concatenation is handled very simply: to simulate the paths in $\alpha \cdot \beta$, starting with a simulation state $\langle S, P \rangle$, first simulate α from $\langle S, P \rangle$ to obtain a simulation state $\langle S', P \wedge P_\alpha \rangle$, then simulate β from $(S', P \wedge P_\alpha)$ to obtain a symbolic state $\langle S'', P \wedge P_\alpha \wedge P_\beta \rangle$.

Union There are two approaches used to simulate $\alpha + \beta$ from pair $\langle S, P \rangle$. The most obvious approach is to simulate α and β separately, yielding two symbolic states. Further simulation would be performed from these states separately. An invariant could be proved by collecting the *set* of symbolic states after an entire simulation, and checking the invariant for each state in the set. The problem with this approach is that it may redo the same work many times, because the symbolic states will be very similar.

The second approach is to compute the *merge* of the two end states. This approximation seems crude, but (surprisingly), works very well for verifying the TORCH Instruction Fetch Unit, described in Section 4, and greatly reduces the complexity of verification. However, note that this may lead to false errors in other designs.

Currently, the choice of which method to use for unions is manual. The regular expression is split into separate expressions which are simulated separately to give several pairs. Pairs are merged for union operations within the individual expressions.

Repetition Obviously, one of the major problems is that there are an infinite number of paths when the regular expression contains stars. The star operator is handled by merging the results of simulating the starred expression a small number of times, to find a symbolic state that subsumes the symbolic states that would be computed by exactly simulating all possible numbers of iterations. This approach is similar to that used with MDGs by Zhou et al. [26].

The basic method is to repeatedly simulate the expression inside the loop, merging the result with the previous result until the symbolic state is identical to the result from the previous iteration, modulo renaming of the fresh variables. The pair $\langle S^*, P^* \rangle$ resulting from this process approximates all the pairs that would have resulted from simulating each of the paths represented by the cycle.

In some cases, this loop generalization is too conservative, resulting in false negatives. For instance, in the TORCH memory system design some pipelined registers take several cycles to reach a stable state when the FSM traverses the loop. If these registers are generalized early, the method will propagate the fresh variables to other state variables, causing them to be generalized unnecessarily. To get a better loop approximation, the user may direct the simulator to traverse the loop several times before the simulator performs the generalization.

3 Verification Process

The the verification process is illustrated in Figure 3. Given a description of the implementation in synthesizable Verilog, a translator converts the Verilog description into an HLN description. A HLN machine corresponds to a Verilog module and consists of a set of input names, output names, a set of type declarations which defines all input, output, and local variables, and a set of behavior descriptions.

The HLN description is fed to an interactive guidance tool. The guidance tool takes the HLN and the regular expression extracted from the FSM. It then simulates along the paths described by the regular expression, with user interaction to determine how to merge states in if-then-else constructs and around loops. Before the simulation, a symbolic state and a symbolic input set are created. The symbolic simulator then uses this state, input set, and the next state transition function to repeatedly execute the implementation and produce the next symbolic state. The simulator in turn calls a decision procedure which is used to simplify the state and used to check equality between states of separate execution paths. The end result is a set of simulation states, one for each simulation path.

A proof obligation is then created for each simulation state. Let $\langle S_i', P_i' \rangle$ be the simulation state resulting from simulation path number i, starting from $\langle S, \mathit{True} \rangle$. The proof obligation to prove correctness invariant I is then:

$$(P_i' \wedge I(S)) \Rightarrow I(S_i')$$

If the proof obligations generated for all simulation paths are valid and I is also true in the initial state, then I is an invariant in all reachable clean states.

The logical formula is then fed to SVC (the Stanford Validity Checker) for checking. SVC is a decision procedure for quantifier-free first-order logic and uses an algorithm similar to the algorithms by Shostak [20, 19] and Nelson-Oppen [17]. The input Boolean formula to SVC can contain Boolean operators, uninterpreted functions and interpreted functions, and distinct constants such as the Boolean truth and bit constants.

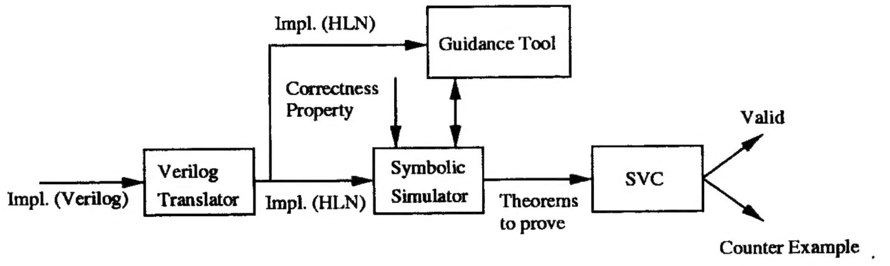

Fig. 3. The verification process.

It may also include the finite bitvectors and records used to model the state of the hardware. SVC will either return with "Valid", or a counter-example. As SVC is used here, the former indicates that the invariant holds, and the latter indicates that the invariant is wrong, or one of the approximation steps has lost a critical constraint.

4 TORCH

We have applied the approach to verifying the correctness of the Instruction Fetch Unit in the TORCH microprocessor. The TORCH design was created by Horowitz's group at Stanford University from 1991-1992 and later optimized. It was constructed for research into microprocessor architectures and has not been fabricated. TORCH is an extension of the MIPS R2000/3000 design [12], which is a 32 bit instruction architecture with a five stage pipeline. It has been simulated (nonsymbolically) extensively, although not to the same degree as in an industrial setting where the resources available for simulation are far greater. The Verilog source code of the TORCH is publicly available at http://www-flash.stanford.edu/torch/.

The TORCH architecture is sketched in Figure 4. TORCH extends the MIPS architecture with some extra optimizing features. It includes *two* asymmetric pipelines with dual issue and dual retirement. To hold various status bits introduced by compiler optimizations, the 32 bit instructions are extended with an extra byte, making the instructions 40 bit wide. For debugging purposes, TORCH can run in a special MIPS compatible mode, where the optimizations are turned off (and the extra byte ignored).

We have previously verified properties of the RTL description of the Instruction Fetch Unit (IFU) [23]. The IFU consists of four modules: an instruction cache (ICache), the IFetch Data Path, the IFetch Control, and the PC Unit Data Path. The description of the IFU consists of 1700 lines of Verilog and uses various common library routines that total approximately 300 additional lines. There are 60 bits of control state (including state machines and random registers with control information); 566 bits of the PC and saved PCs/next PCs, etc.; 1200 bits of explicit data registers. The cache and memory are modelled as unbounded arrays, but each memory word is 64 bits wide and each cache line is 320 bits wide. In addition, each cache line has a 24 bit tag. A block diagram appears in Figure 5.

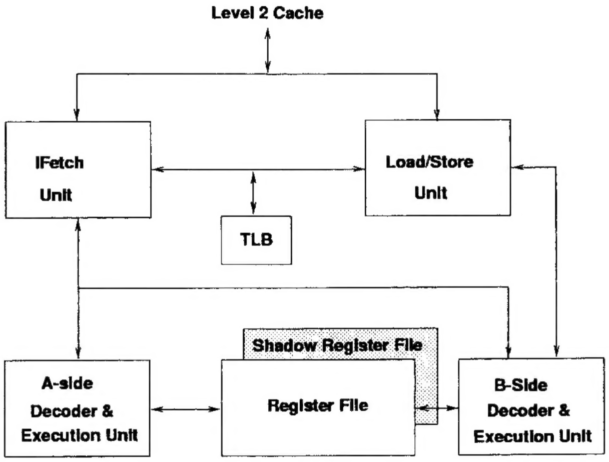

Fig. 4. A diagrammatic overview of the TORCH architecture.

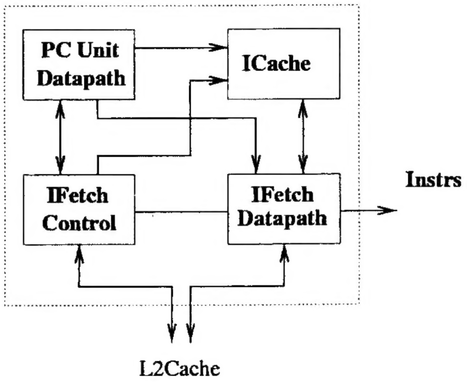

Fig. 5. The Instruction Fetch Unit (IFU).

The PC Unit Data Path maintains a program counter (PC) and calculates the next PC based on input from the surrounding modules, in particular the decode/execute module. The IFetch Data Path and Control return the instruction corresponding to a given PC. The PC is looked up in the ICache by matching the PC with the ICache tags. If there is a match, called a *hit*, the instruction is returned. Otherwise, there is a *miss* and the IFetch Control will output a stall signal and initiate communication with the main memory (in actuality, the ICache communicates with a level 2 cache, but the model here merges the level 2 cache and memory into a single unit) to fill the 8 instruction cache lines (8 $*$ (32 MIPS bits $+$ 8 bits status information) $=$ 320 bits). Once its request is being serviced, the IFU receives two 32 bit words per cycle in 5 cycles, in total 10 words. The first two words contain the eight status bytes, and the last 8 words are the MIPS instructions. Each instruction is matched with its status byte and stored in the cache line. The cache has 1,024 cache lines. Following this, a *refetch* occurs, the stall signal is lowered, and the instruction is provided on the interface.

The unpacking and matching of MIPS instructions with status bytes is carried out to provide compatibility with the MIPS architecture. Information is stored in 32 bit words in memory. The unpacking is of course turned off when TORCH runs in MIPS mode and only 8 words are loaded from memory during a cache miss.

5 Verifying TORCH

The correctness invariant is:

> *For every instruction location, if the location is registered in a valid cache line, the contents of the cache line are the same as the contents of the line in memory.*

The wording of this invariant is intentionally vague. "Contents of" hides the fact that the true invariant is almost a page long, because the format of the data in the cache is different from the format in memory. As the data is transferred, an extra byte of status information is appended to every 32 bit word in the cache. So, comparing the contents of the cache and memory requires extracting the appropriate 32 bit fields before comparing them with the memory. Writing this expression is a bit tedious, but not intellectually difficult.

There is an FSM in the IFetch control logic and also one in the memory module. The two state machines together implement the data transfer protocol on each side of the memory bus. Each controls the data transfer on its side and keeps track of the progress of the current transfer the memory line, i.e., how many pairs of instructions that have been transferred. We form the product of the two FSMs to form a single FSM representing the data transfer. The TORCH mode FSM is illustrated in Figure 6. The FSM for the MIPS mode is similar, but slightly simpler.

The input signals of the FSM are: (1) cache miss (M) which triggers the cache miss process, (2) MIPS mode (P) or TORCH mode $\overline{P}$, (3) the ITLB miss (T) which signals a miss in the TLB, (4) a nondeterministic delay (D) modeling the delay in the main memory or level two cache, and (5) level two cache miss (L). The cache miss M is a predicate on several state variables, and all other signals are inputs to the memory system. The set of input symbols of the FSM is the set of predicates which consists of terms from the following set:

$$\{M, \overline{M}, \overline{M}, P, \overline{P}, T, \overline{T}, L, \overline{L}, D, \overline{D}\}$$

The regular expression corresponding to the FSM is a union of three major paths (see Figure 7). The first path in the expression represents a cache hit. The other two represent a cache miss with and without TLB miss, respectively.

The memory is an array and the cache is an array of records containing three fields: the valid bit (***valid***), the cacheline data (***data***), and the memory address (***addr***) of this

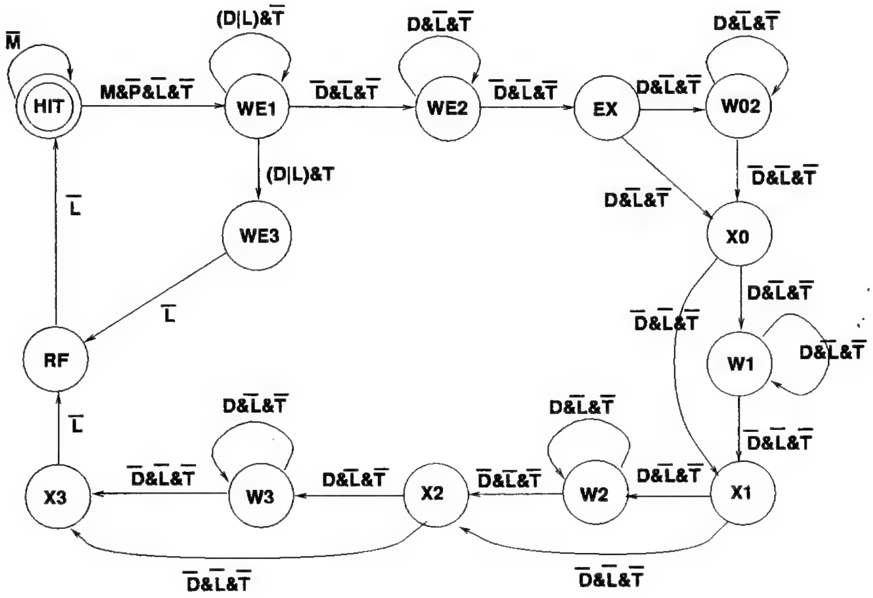
$\overline{M}$
HIT
$M\&\overline{P}\&\overline{L}\&\overline{T}$
$(D|L)\&\overline{T}$
WE1
$\overline{D}\&\overline{L}\&\overline{T}$
$D\&\overline{L}\&\overline{T}$
WE2
$\overline{D}\&\overline{L}\&\overline{T}$
EX
$D\&\overline{L}\&\overline{T}$
$D\&\overline{L}\&\overline{T}$
W02
$(D|L)\&T$
WE3
$\overline{D}\&\overline{L}\&\overline{T}$
$D\&\overline{L}\&\overline{T}$
X0
$\overline{L}$
$\overline{L}$
RF
$D\&\overline{L}\&\overline{T}$
$\overline{D}\&\overline{L}\&\overline{T}$
W1
$D\&\overline{L}\&\overline{T}$
$\overline{D}\&\overline{L}\&\overline{T}$
$\overline{L}$
$D\&\overline{L}\&\overline{T}$
$D\&\overline{L}\&\overline{T}$
$\overline{D}\&\overline{L}\&\overline{T}$
X3
W3
$D\&\overline{L}\&\overline{T}$
X2
$\overline{D}\&\overline{L}\&\overline{T}$
W2
$D\&\overline{L}\&\overline{T}$
X1
$\overline{D}\&\overline{L}\&\overline{T}$
$\overline{D}\&\overline{L}\&\overline{T}$

Fig. 6. The product of the two memory system FSMs for the TORCH mode. For clarity we have left out $\overline{P}$ (indicating TORCH mode) from every transition.

$$
\begin{aligned}
&[\ \ [\overline{P} \wedge \overline{M}]\\
&+ [(\overline{P} \wedge M \wedge \overline{T}) \cdot (\overline{P} \wedge (D \vee L) \wedge \overline{T})^* \cdot (\overline{P} \wedge (D \vee L) \wedge T) \cdot \overline{P} \cdot (\overline{P} \wedge \overline{L})]\\
&+ [(\overline{P} \wedge M \wedge \overline{T}) \cdot (\overline{P} \wedge (D \vee L) \wedge \overline{T})^* \cdot (\overline{P} \wedge \overline{D} \wedge \overline{L} \wedge \overline{T})\cdot\\
&\quad (\overline{P} \wedge D \wedge \overline{L} \wedge \overline{T})^* \cdot (\overline{P} \wedge \overline{D} \wedge \overline{L} \wedge \overline{T})\cdot\\
&\quad ((\overline{P} \wedge \overline{D} \wedge \overline{L} \wedge \overline{T}) + (\overline{P} \wedge D \wedge \overline{L} \wedge \overline{T}) \cdot (\overline{P} \wedge D \wedge \overline{L} \wedge \overline{T})^* \cdot (\overline{P} \wedge \overline{D} \wedge \overline{L} \wedge \overline{T}))\cdot\\
&\quad ((\overline{P} \wedge \overline{D} \wedge \overline{L} \wedge \overline{T}) + (\overline{P} \wedge D \wedge \overline{L} \wedge \overline{T}) \cdot (\overline{P} \wedge D \wedge \overline{L} \wedge \overline{T})^* \cdot (\overline{P} \wedge \overline{D} \wedge \overline{L} \wedge \overline{T}))\cdot\\
&\quad ((\overline{P} \wedge \overline{D} \wedge \overline{L} \wedge \overline{T}) + (\overline{P} \wedge D \wedge \overline{L} \wedge \overline{T}) \cdot (\overline{P} \wedge D \wedge \overline{L} \wedge \overline{T})^* \cdot (\overline{P} \wedge \overline{D} \wedge \overline{L} \wedge \overline{T}))\cdot\\
&\quad ((\overline{P} \wedge \overline{D} \wedge \overline{L} \wedge \overline{T}) + (\overline{P} \wedge D \wedge \overline{L} \wedge \overline{T}) \cdot (\overline{P} \wedge D \wedge \overline{L} \wedge \overline{T})^* \cdot (\overline{P} \wedge \overline{D} \wedge \overline{L} \wedge \overline{T}))\cdot\\
&\quad (\overline{P} \wedge \overline{L} \wedge \overline{T}) \cdot (\overline{P} \wedge \overline{L})]]
\end{aligned}
$$

Fig. 7. The regular expression corresponding to the TORCH mode FSM.

cacheline. The correctness invariant above (that the contents of valid cache lines are the same as the corresponding location in memory) must be proved. The base case of the induction proof is trivial, since no cache lines are valid (the initial state is the state immediately after initialization, and initialization flushes the cache). For the inductive step, is to prove that if the property holds in an clean state S, then it is also valid in the next clean state S'. A small trick is required to deal with the quantifier, since SVC doesn't support quantification: we substitute a fresh variable for the quantifier in the consequent (this is called *Skolemizing*) and manually instantiate the quantifier in the antecedent as necessary.

A prototype guidance tool has been implemented in Lisp and applied it in the verification of the IFU. In the proof, each of the three paths in the regular expression was handled separately. The proof for MIPS mode was similar. The run-times for simulation and subsequent verification are shown Figure 8. The last column in the table lists the possible number of paths after loop generalization. However, unions of paths were collapsed into a single path using the generalization method described above.

Regular Expr.	Mode	Run-time	Memory usage	Possible Number of paths
1	MIPS	00:17	16 MBytes	1
2	MIPS	03:43	52 MBytes	2
3	MIPS	30:22	72 MBytes	108
1	TORCH	00:17	16MBytes	1
2	TORCH	04:05	53 MBytes	2
3	TORCH	84:40	81 MBytes	324

Fig. 8. Combined run-times of the simulation and verification required for each path.

From the table of Figure 8, paths specified by the third and the sixth regular expressions take much more time than the rest of the paths do. This is due to the cost of simulation. Most of the simulation time is spent on finding the loop approximation and merging paths from union or loops, which needs to compare variables in two states. Since the memory system has 86 state variables, the comparison is quite expensive. However, without merging parallel paths, we would have 108 ($108=2^2 \times 3^3$) and 324 ($324=2^2 \times 3^4$) paths (each of the first two stars generating two paths and each union generating three paths) for MIPS 3 and TORCH 3, respectively. If the simulation had not merged these parallel paths, the time for simulating these paths would have been about 108 and 324 times of the listed time for these two regular expression. The proofs were carried out with many fewer strengthening invariants than previously needed. Apart from the property, 12 extra conjuncts were needed to strengthen the invariant.

Two of these simply list the reachable states of the FSM for the phase one and phase two latches of the state bits of the FSM. These invariants are evident from inspection of the HDL source, or could be computed automatically by well-known state enumeration algorithms. There is another invariant that says that the phase one and phase two latches are the next state and current state of the FSM. This invariant is also apparent from the HDL, or could be computed by extracting the next state function. There is also a simple invariant that says that the internal reset low when the input reset is low.

Four of these were simple historyless invariants and were found by manually applying and existing method [24]. Here is a typical one: $\overline{MemStallS1} \Rightarrow (FSMS1 = FSMS2)$. This invariant says that whenever *MemStallS1* is false, the phase two variable (*FSMS2*) has been overwritten by the phase one variable (*FSMS1*).

Four of the conjuncts (see Figure 9) had to be found by trial and error. Some of these may be historyless properties that could not be found by the (incomplete) technique that was used. There was one unexpected issue in the design that may contribute to the need for these invariants. The FSM does not discover a read miss until the following cycle, when it transitions to reflect the cache miss. The *Hit* state in the FSM thus corresponds both to the hit states in the data path and the first miss state. Since the clean states are

$TagS2 = PCChainS2r[29:3]$
$(FSMS2 = HIT) \land IStallS2 \Rightarrow (PC = PCChainS1r)$
$(FSMS2 = HIT) \land \overline{IStallS2} \land TORCHMODE \Rightarrow$ $(MemAddress = 5 \times PCChaincs2r) \land (MemAddress = 5 \times PCChainS1r)$
$(((FSMS2 = HIT) \land ICacheMiss) \lor FSMS2 \neq HIT) \land TORCHMODE$ $\Rightarrow (MemAddress = 5 \times PCChainS2r)$

Fig. 9. Four conjuncts of invariants found by inspection and trial and error. These are logically accurate, but the notation and variable names have been modified for readability.

defined as the *hit* states, this requires the reachability invariant to be inductive on both kinds of states.

This effort uncovered the same bugs as found by the previous method [23]. The first bug is that it writes the tag of a noncacheable instruction into the ICache tag register file. As in the MIPS architecture, TORCH provides for both cacheable and noncacheable memory accesses. When the IFU fetches a noncacheable instruction, it causes a cache miss and sends the request to main memory. When the noncacheable data arrives at the IFU, the requested instruction is passed on to the decode/execute unit. Since the instruction is noncacheable, neither the data nor the tag should be written to the ICache. However, a bug in the implementation of the IFU causes the address tag of noncacheable instructions to be written into the ICache tag, while the noncacheable data is not.

The other two bugs were found from the control logic of the PC tag. The first bug causes the PC tag latch to not keep the current PC tag during an ICache miss. Instead, it incorrectly stores the tag of the immediately following instruction. At the end of the ICache miss, the tag corresponding to the next instruction is written into the ICache tag register file while the ICache data corresponding to the current PC is correctly written as ICache data.

We discovered a second bug immediately after the first one was corrected. Consider two branches B_1 and B_2 with target addresses that have different tags T_1 and T_2 but identical cache line indices. The bug manifests itself at the end of the ICache miss caused by the target T_1 of B_1: After the ICache line has been updated, the IFU issues an internal refetch command to the ICache. This refetch gets the designated instructions and its tag from the ICache, and the ICache tag is compared with the PC tag as in a normal fetch. However, the bug causes the next PC tag to be loaded into the PC tag latch before this comparison is done. Thus, if the new PC tag is different from the old PC tag, this refetch will generate another ICache miss. This second cache miss will cause the same data to be fetched, but will wrongly store target address T_2 of B_2 in the tag register file. Following this, the instruction for the target of B_1 is correctly returned. However, when the instruction for the target T_2 of B_2 is requested, T_2 is already in the target register file, which wrongly causes a hit. As a result, an instruction from the B_1 cacheline is incorrectly returned. As before, this causes TORCH to following a wrong path of execution.

The original effort took 2 person-months. It is hard to estimate the effort required by the new method, since the prototype tools were being developed while doing the verification. Furthermore, the user was familiar with the design of TORCH as well as the invariants. We guess that it would have taken 1-2 person-weeks without previous knowledge of TORCH and the invariants.

6 Discussion

The technique described here is not universal. There are several restrictions and assumptions about the design style:

- There is a simple property to be proved on some subset of the states (which we will designate as the clean states).
- The number of clean states is small.
- The product of the control FSMs is small (no more than hundreds of states).
- Cycles are wait loops, in which state variables do not change.
- The control is not pipelined.

Interestingly, at least some designs fall within these guidelines, yet state enumeration approaches are not easy to apply because of the difficulty of verifying the control FSMs independently of the data path. We are investigating ways to remove and generalize these restrictions.

The method needs to be more automated. The current implementation still involves some manual work, such as extracting the finite state machine and finding the regular expression. From the design in the HDL description, it should be possible to generate the transition function for the control state machine automatically and then generate the regular expressions using the existing algorithm. In addition, there needs to be a tool that finds the historyless invariants automatically.

Acknowledgements

This work was sponsored under contract numbers DABT63-95-C-0049-P00005 and DABT63-96-C-0097-P00002. The contents of this paper do not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.

References

1. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification (CAV)98*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331, Vancouver, BC, Canada, June/July 1998. Springer-Verlag.
2. Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification (CAV)96*, volume 1102 of *Lecture Notes in Computer Science*, pages 323–335, New Brunswick, NJ, July/August 1996. Springer-Verlag.
3. N. S. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, February 1997.

4. R. E. Bryant, D. L. Beatty, and C.-J. H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *28th ACM/IEEE Design Automation Conference*, 1991.
5. J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *28th ACM/IEEE Design Automation Conference*, 1991.
6. J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, 1990.
7. M. Caplain. Finding invariant assertions for proving programs. In *International Conference on Reliable Software*, pages 165–171, 1975.
8. W. Carter, W. Joyner, and D. Brand. Symbolic simulation for correct machine design. In *16th Design Automation Conference Proceedings (1979)*, pages 280–286, June 1979.
9. R. Floyd. Assigning meaning to programs. In *Proc. Symposium in Applied Mathematics*, volume 19, pages 19–32, 1967.
10. R. W. Floyd. *The Language of machines: an introduction to computability and formal languages*. New York : Computer Science Press, 1994.
11. S. German and B. Wegbreit. A synthesizer of inductive assertions. *IEEE Transactions on Software Enginnering*, 1(1):68–75, March 1975.
12. G. Kane. *MIPS RISC Architecture*. Prentice Hall, 1988.
13. S. Katz and Z. Manna. A heuristic approach to program verification. In *Proceedings: 3rd International Joint Conference on Artificial Intelligence*, pages 500–512, 1976.
14. J King. A program verifier. In *Information Processing 71 Proceedings of the IFIP Congress*, volume 1, pages 234–249, 1972.
15. B. Levy. Microcode verification using sdvs-the method and a case study. In *17th MICRO (1984)*, pages 234–245, 1984.
16. R. Mueller and M. Ruda. Formal methods of microcode verification and synthesis. *IEEE Software*, 3(4):38–48, July 1986.
17. G.E. Nelson and D.C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.
18. Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer Aided Verification (CAV)97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.
19. R.E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, April 1979.
20. R.E. Shostak. Deciding combinations of theories. Technical Report SRI-CSL-132, Computer Science Laboratory, SRI International, February 1982.
21. M. Smith, M. Horowitz, and M. Lam. Efficient superscalar performance through boosting. In *5th International Conference on Architectural Support for Programming languages and Operating Systems*, pages 248–259, Boston, MA, 1992. IEEE/ACM.
22. M. Smith, M. Lam, and M. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *17th International Symposium on Computer Architecture*, volume 18-2, pages 344–354, Seattle, WA, May 1990. IEEE/ACM.
23. J. Su, L. Arditi, S. Das, J. U. Skakkebæk, and D. L. Dill. Formal verification of the TORCH microprocessor RTL design. Unpublished, 1998.
24. Jeffrey X. Su, David L. Dill, and Clark W. Barrett. Automatic generation of invariants in processor verification. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer Aided Design (FMCAD)*, volume 1166 of *Lecture Notes in Computer Science*, pages 197–201. Springer-Verlag, November 1996.
25. R. Tarjan. A unify approach to path problems. *Journal of the ACM*, 28(3):577–593, July 1981.
26. Z. Zhou, X. Song, S. Tahar, E. Cerny, F. Corella, and M. Langevin. Formal verification of the island tunnel controller using multiway decision graphs. In M. Srivas and A. Camilleri,

editors, *Formal Methods in Computer Aided Design (FMCAD)*, volume 1166, pages 233–247. Springer-Verlag, November 1996.